

# Greedy Algorithms

## SCS – Data Structures and Algorithms III Assignment - Report

Greedy Algorithm



## Question 01

### problem

Sandun is a wine lover. He had to be in his home for three months due to the Covid-19 situation. Sandun wants to buy some wine since the curfew has been lifted. He goes to his favorite wine yard with some empty bottles. Wine yard owner is pleased to see Sandun. Hence, he offers Sandun free wine on one condition. Sandun can fill only the bottles he carries in his bag.

Wine barrels in the wine yard have a tag in each one of them which has two values containing the number of bottles in the barrel and the price of the whole barrel.

You have to build a program to help Sandun to take maximum valued wine from this offer.

Given the number of bottles that he brought and the information on the wine barrels, you have to find the maximum value of wines Sandun gets from this deal.

Please note that he cannot fill any of the bottles with more than one type of wine. Assume that the bottle is a 750ml glass bottle.

Conditions for filling wine to bottles :

- Person can only fill the bottles which are in his bag.
- When he fills the wine bottles, he can't mix the wine.

### Task

Find the maximum value of wine bottles that he can carry from his bag. So this is a **maximization problem**.

### Algorithm

Let's take an example for better explanation.

Input x[ ] //Number of bottles and number of barrels

**//candidate set**

x[0]=Number of bottles      x[1]=Number of barrels

Input y[ ] //Volumes of the barrels in the bottles.

Input z[ ] //Price of each bottle.

Get count to keep the types of wine.

Initialize p[ ] // array with decision variable

For i=1 to count do

    p[i]=z[i]/y[i]      //p array store cost per bottle  
                             //fraction variable

price\_v = 0; //maximum value of

Then call the price function

maxArray = max(p); // **selection function**

```
if( y[ maxArray[0] ] > bottles) //feasibility function
    price_v=bottles*p[ maxArray[0] ]
else
    price_v=y[ maxArray[0] ] * p[maxArray[0]]
    //objective function
    bottles=bottles- y[ maxArray[0] ] //remain required bot
    P[ maxArray[0] ]=0 //previous max position equal to 0
    (check with out previous position)
    price_v=price_v+price( )
return price_v; //solution function
}
```

Complexity = $O(n \log n)$

For describing the algorithms let's take an example

Number of bottles that the person have= 5

Number of barrels=3

Volume	1	2	3
Price of barrel	6	10	12
Average Cost	6	5	4

By using this algorithm , we can modify the table.

Price of barrel	6	10	4
Average Cost	6	5	4
Balance Volume	0	0	1

First 1 volume get 1 bottles

Next 2 volume get 2 bottles

Next 3 volume get 2 bottles

5 bottles

So maximum price=  $1*6+ 2*5+ 2*4=24$

## Result of the Algorithm

## Easy scenario

```
Output - DSA (run) %
run:
Number of Bottles & Number of barrels :5 3
Volume of the barrels in bottles :1 2 3
Price of each barrel :5 10 12
The value is 24.0
BUILD SUCCESSFUL (total time: 19 seconds)
```

## Complex scenario

```
Output - DSA (run) %
run:
Number of Bottles & Number of barrels :4 4
Volume of the barrels in bottles :1.5 1 1 0.5
Price of each barrel :150 60 55 10
The value is 265.0
BUILD SUCCESSFUL (total time: 1 minute 1 second)
```

```
Output - DSA (run) %
run:
Number of Bottles & Number of barrels :4 4
Volume of the barrels in bottles :2 1.2 1 0.8
Price of each barrel :200 72 50 40
The value is 310.0
BUILD SUCCESSFUL (total time: 1 minute 16 seconds)
```

```
Output - DSA (run) %  
run:  
Number of Bottles & Number of barrels :4 3  
Volume of the barrels in bottles :2.5 3 1  
Price of each barrel :12.5 12 6  
The value is 20.0  
BUILD SUCCESSFUL (total time: 25 seconds)
```

```
Output - DSA (run) %  
run:  
Number of Bottles & Number of barrels :7 4  
Volume of the barrels in bottles :4.5 2 0.5 1  
Price of each barrel :15 16 5 6  
The value is 62.0  
BUILD SUCCESSFUL (total time: 56 seconds)
```

## Question 02

### Problem

Manel is a school teacher. She wants to give some face masks to the students in her class. All the students sit in a line and each of them has a score according to his or her performance in the class. Students can't change their seating order. Manel wants to give at least 1 face mask to each student. If two students sit next to each other, then the one with the higher score must get more face masks. Note that when two children have equal scores, they are allowed to have a different number of face masks. Manel wants to minimize the total number of face masks she must buy.

### Analysis of the problem

This question has certain conditions to when giving the masks to students,

- Every student must receive at least one mask.: So, we can say In the best-case scenario, which is that every student has equal marks, Manel should buy  $n$  number of masks. ( $n$ =number of students)

Best case:

Marks of students	x	x	x	x	x	x	x	.....	x
Number of masks received	1	1	1	1	1	1	1	.....	1

- When two students are sitting next to each other, one with higher marks should receive more masks than the other.: In the worst-case scenario, in which each student has distinct marks, and they sit in order by their marks, Manel should buy  $n(n+1)/2$  number of masks. ( $n$ =number of students)

Worse case: where  $x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < \dots < x_n$ . This table shows if the students sit ascending order according to their marks. Also, it can be other way round if they sit in the descending order of their marks, which gives the same sum of masks  $n(n+1)/2$ .

Marks of students	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	.....	$x_n$
Number of masks received	1	2	3	4	5	6	7	.....	$n$

- When two students are sitting next to each other, and both students have equal marks they can receive different amounts of masks.
- Students' seating order can't be changed.
- Manel wants to minimize the number of masks she should buy.: This is a minimisation problem. The result must be the optimum solution.

According to first two points, we can say our answer is  $n \leq (\text{answer}) \leq n(n+1)/2$ , where  $n$  is the number of students. To calculate the number of masks to buy, we receive the number of students and their marks according to the order they sit.

Initial data:

- Number of students ( $n$ ): Given that  $1 \leq n \leq 10^5$ . We can use integer to save this value.

- Marks (**score[n]**): We can save marks of the students in an integer array as we have given that  $1 \leq \text{score}[i] \leq 10^5$ . Also, the array size should be equal to the number of students. And the seating position of a student is easier to get by the array index.

Resulted data:

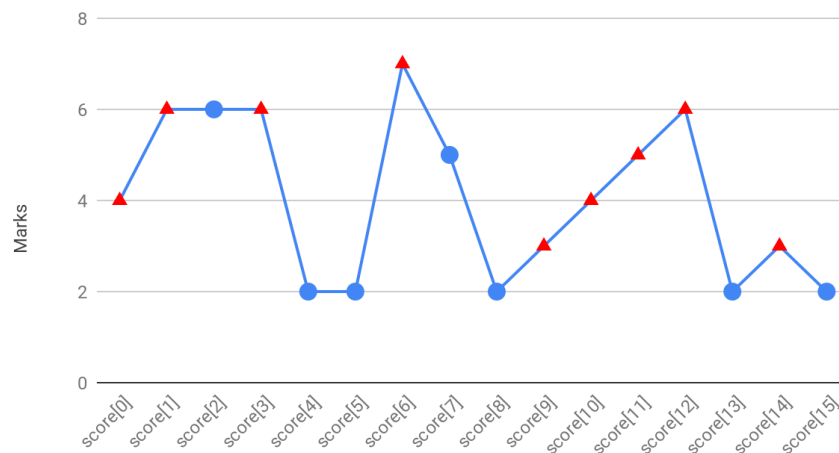
- Minimum number of masks Manel should buy (**n\_mask**): As our worse case scenario gives  $n(n+1)/2$  as the answer, the highest value we receive is  $(10^5(10^5+1))/2$  which sums up to 5000050000. So, we can initialize that value as *long long int*.

## Solution

We keep two more important values to proceed with the solution. Below gives how the variable changes and later we'll look into how these values help to take decisions in our algorithm.

- p - This variable saves a position of a student. Initially it was 0, which is the beginning of the score array. The position which is set to p is the summit value of marks by far. If our `score[n]` is {4, 6, 6, 6, 2, 2, 7, 5, 2, 3, 4, 5, 6, 2, 3, 2} p value moves as 0->1->3->6->9->10->11->12->14. Let's represent this array in a chart.

Marks of student as per position



As you can see p position saves the top indexes and if the marks are climbing p position also climbs.

- pre - This value saves the previously added value to the n\_mask. And this value is immediately set to 0 when a flat position or relatively low position encounters (in chart representation).

To begin calculation, initially we know each student receives one mask, so our starting value of the `n_mask=n` (In the beginning Manel gives one mask to every student). Then observe each student's marks and modify our `n_mask` value according to the situation.

- As the first student has no one to compare before him/her we compare with the second student. If the second student has a high mark or equal mark, Manel doesn't have to give another mask to the first student. And if the first student has a high mark, Manel gives one mask to the first student. (`n_mask=n_mask+1`).

- Then we are moving to the middle of the row, where each student has two students next to him/her. That is each  $\text{score}[i]$  we consider we have a  $\text{score}[i-1]$  and  $\text{score}[i+1]$ .
  - If the  $\text{score}[i-1]$  is less than  $\text{score}[i]$ , either we are climbing or we are in a summit (in chart). That is decided by the next value which is  $\text{score}[i+1]$ . So we change the  $p$  value to  $i$ . As the previous score is lower Manel has to give 1 mask additional to the  $i^{\text{th}}$  student. If  $(i-1)^{\text{th}}$  student has only 1 mask which was given initially, Manel gives a 1 mask to the  $i^{\text{th}}$  student, and the total number of masks he/she has sums up to 2. If he has given an additional mask to the  $(i-1)^{\text{th}}$  student (The total he has is 2). Manel has to give two additional masks to the  $i^{\text{th}}$  student, and the total he has sums up to 3. Likewise, we give 1 mask more than the previous student. As the pre value saves the previously given amount, we can conclude that we give  $\text{pre}+1$  masks to the  $i^{\text{th}}$  student. And our pre value also gets updated to  $\text{pre}+1$  for the future calculations.
  - If the  $\text{score}[i-1]$  is equal to  $\text{score}[i]$ , either we are in a summit or we are or in a flat position of in a lower place (in chart). If we are in a flat position or lower place, that is  $\text{score}[i+1]$  is greater than or equal to  $\text{score}[i]$ , Manel don't have to give a mask to the  $i^{\text{th}}$  student. And if it is a summit where the next student has a lower score, Manel has to give one mask and make the total mask to 2 of the  $i^{\text{th}}$  student as  $(i+1)^{\text{th}}$  student already has 1 mask. And it sets our pre value to 1 as Manel gave 1 mask. Also, we have to move the  $p$  to the summit position.
  - Then, if the  $\text{score}[i]$  is lower than  $\text{score}[i-1]$ , either we are in a slope or in a lower place (in chart). If we are in a lower place where the  $\text{score}[i+1]$  is greater than  $\text{score}[i]$ , we don't have to give any mask to the  $i^{\text{th}}$  student. But if we are in a slope then, Manel has to give 1 mask to  $i^{\text{th}}$  student to deal with the  $(i+1)^{\text{th}}$  student, and give another mask to  $(i-1)^{\text{th}}$  student to deal with  $i^{\text{th}}$  student and so on. As Manel has to give 1 mask additional to every student going back the row, this is where she can use the  $i$  and pre value to decide the total mask to be allocate in this position. Let's see it with a example,

	$i=0$ $p=0$	$i=1$	$i=2$	$i=3$
Marks	6	5	4	3
Initially we give one mask	1	1	1	1
Move to 1st student <b>(+1)</b> $\text{pre}=1$ (as we gave 1)	2 (+1)	1	1	1
Move to 2nd student <b>(+2)</b>	3 (+1)	2 (+1)	1	1
Move to 3rd student <b>(+3)</b>	4 (+1)	3 (+1)	2 (+1)	1

As you can see the value we add at each position is given by  $(i-p)+1$ .

And if we encounter a situation like below,

	....	$i=n$ $p=n$	$i=n+1$	$i=n+2$	$i=n+3$
--	------	----------------	---------	---------	---------



Marks	....	6	5	4	3
$n^{\text{th}}$ encounter, we have given 3 additional masks pre=3(as we gave 3)	....	4	1	1	1
Move to $n+1$ student <b>(+1)</b> pre=1(as we gave 1)	....	4	2 (+1)	1	1
Move to $n+2$ student <b>(+2)</b>	....	4	3 (+1)	2 (+1)	1

As you can see the value we add at each position is  $(i-p)$ . This happens when our pre value is greater than the position gap to summit which is  $(i-p)$ . In our previous scenario pre value is less than position gap  $(i-p)$ .

- Then, when we arrive at the end of the row, we only consider the previous score. If the previous student's score is lower we give one additional mask. If not we can ignore it.

After going through the entire row by considering relevant scenario at each position, our  $n\_mask$  value finally sums up to the minimum number of masks Manel should buy.

Algorithm

Int  $n$  // number of students

Int  $score[n]$  // marks of student **(Candidate set)**

long long int  $n\_mask=n$ ; // initial value of number of mask is  $n$

For 0 to  $n-1$ {

```

    if( $i==0$  &&  $score[i]>score[i+1]$ ){ //check beginning of the array (Feasibility function)
         $n\_mask=n\_mask+1$ ;  //(Objective function)
         $pre=1$ ;
    }

```

```

    else if( $i==n-1$  &&  $score[i]>score[i-1]$ ){ //check end of the array (Feasibility function)
         $n\_mask=n\_mask+pre+1$ ;  //(Objective function)
    }

```

```

    else if( $score[i-1]<score[i]$ ){ // previous student's score is less (Feasibility function)
         $p=i$ ;
         $n\_mask=n\_mask+pre+1$ ;  //(Objective function)
         $pre=pre+1$ ;
    }

```

```

    else if( $score[i-1]==score[i]$ ){ // previous student's score is equal (Feasibility function)
        if( $score[i]>score[i+1]$ ){  //(Selection function)
             $p=i$ ;
             $n\_mask=n\_mask+1$ ;  //(Objective function)
        }
    }
}

```

```

        pre=1;
    }
    else{
        pre=0;
    }
}

else if(score[i-1]>score[i]){ // previous student's score is higher (Feasibility function)
    if(score[i]>score[i+1]){ //(Selection function)
        if(pre>(i-p)){ //(Selection function)
            n_mask=n_mask+(i-p); //(Objective function)
        }
        else{
            n_mask=n_mask+(i-p)+1; //(Objective function)
        }
    }
    else{
        pre=0;
    }
}
}
}

```

Print n\_mask; //output the minimum number of masks Manel should buy **(Solution function)**

Workout example

n=3

Student's position	0	1	2
Marks	1	2	2
n_mask=n=3	1	1	1
0th iterate   p=0 pre=0   n_mask=3	1	1	1
1st iterate   p=1 pre=1   n_mask=4	1	2 (+1)	1
2nd iterate   p=1 pre=1   n_mask=4	1	2	1

Result of the algorithm

- $n=3$  score[n]={1,2,2}

```
3
1
2
2

4
-----
Process exited after 7.093 seconds with return value 0
Press any key to continue . . .
```

- $n=6$  score[n]={4,6,4,5,6,2}

```
6
4
6
4
5
6
2

10
-----
Process exited after 40.12 seconds with return value 0
Press any key to continue . . .
```

n=10 score[n]={6,6,6,6,6,6,6,6,6,6} best case

```
10
6
6
6
6
6
6
6
6
6
6
6
10
-----
Process exited after 23.41 seconds with return value 0
Press any key to continue . . .
```

n=8 score[n]={15,14,13,12,11,10,9,8} worse case

```
8
15
14
13
12
11
10
9
8
36
-----
Process exited after 27.94 seconds with return value 0
Press any key to continue . . .
```

## Question 03

### Problem

Worker works for an online store. Her task is to determine the lowest cost way to combine her orders for shipping. She has a list of product weights. The shipping company has a requirement that all products loaded in a container must weigh less than or equal to 4 units plus the weight of the minimum weight product. Otherwise they will not take responsibility for addresses getting defaced. All products meeting that requirement will be shipped in one container.

After storing the list of products with weights as an array we have to sort that. Then, we can start at the beginning of the sorted weight array. Then after the sorting weight list, we can pass that array into our function.

Firstly take index 0 first and get a value as `arr[0]`. Then, the loop is started and checks the weights of the products. If there is a weight which is less than `arr[0]+4` weight it doesn't matter. But if not, the loop will stop and split the previous checked weighted into one container and after that if it is not the end of the array, the algorithm initializes a new index. then loops goes like previous. according to the this way we can find the required number of containers

### Algorithm

Let's take an example for better explanation.

Suppose there are 8 number of products {1,2,3,21,7,12,14,21} (**candidate set**)

Firstly, those weighted lists are inserted into the array.

Index	0	1	2	3	4	5	6	7
weight	1	2	3	21	7	12	14	21

Then, sort the weighted array ascending order.

Index	0	1	2	3	4	5	6	7
sorted weight	1	2	3	7	12	14	21	21

{1,2,3,7,12,14,21,21} (**selection function**)

After the sorting array, pass to the function.

For( `i=0 to n` ) // `n` = number of shipping products

Check the constraints and if it is pass then do:

```
k = arr[i] + 4
```

```
While (arr[i] <= k) // arr[i] is the greedy choice (feasibility function)
```

```
{ i++ ; }
```

If it is not the end of the array:

```
i = i - 1;
```

```
Do: count + 1; (objective function)
```

If find the required number of container:

```
Int count =0;
```

```
For ( i=0 to n)
```

```
count ++
```

```
return count; (solution function)
```

According to the above example we can see the number of containers we want and how to divide the list of weights as follows.

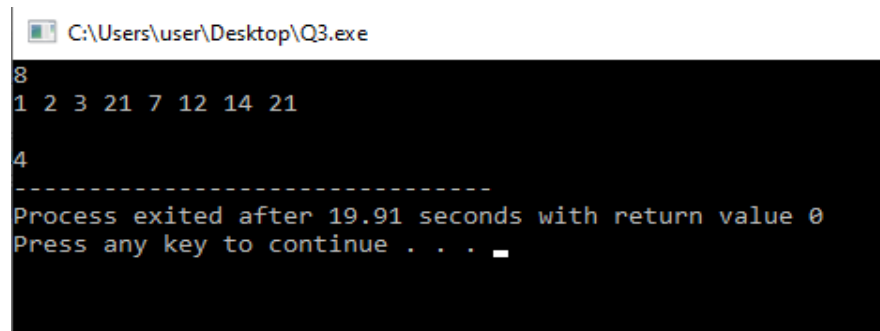
1st container => 1,2,3

2nd container => 7

3rd container => 12,14

4th container => 21,21

Results of the algorithm



```
C:\Users\user\Desktop\Q3.exe
8
1 2 3 21 7 12 14 21
4
-----
Process exited after 19.91 seconds with return value 0
Press any key to continue . . .
```

## Team Members – Group 52

Name	Index Number
N.K.J.Sandunika	18001505
G.R.N.Sankalani	18001513
C.D.Satharasinghe	18001521